ReXSim: A Retargetable Framework for Instruction-Set Architecture Simulation

Mehrdad Reshadi, Prabhat Mishra, Nikhil Bansal, Nikil Dutt

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine.
{reshadi, pmishra, nbansal, dutt}@cecs.uci.edu
http://www.cecs.uci.edu/~aces

CECS Technical Report #03-05
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA
February, 2003

Abstract

Instruction-set simulators are an integral part of today's processor and software design process. Due to increasing complexity of the architectures and time-to-market pressure, performance and retargetability are the most important features of an instruction-set simulator. Dynamic behavior of applications and processors requires the ISA simulators to be flexible. Flexible interpretive simulators are slow while fast compiled simulators are not flexible enough. Retargetability and flexibility require generic models while high performance demands target specific customizations. To address these contradictory requirements, we propose a generic model as well as an efficient and flexible implementation technique. The contribution of this paper is a simulation framework that is retargetable, fast and flexible. We have developed a generic instruction model and a generic decode algorithm to generate retargetable simulators that supports wide spectrum of processor architectures including RISC, DSP, VLIW and Superscalar. We have also developed the Instruction-Set Compiled Simulation (IS-CS) technique that combines the performance of compiled simulation with the flexibility of interpretive simulation. The generated simulator delivers up to 46% performance improvement over JIT-CCS [2], the best known result in this category. We illustrate the applicability of our approach using two different state-of-the-art real world architectures: the Sparc and the ARM.

Contents

List of Figures

List of Tables

1 Introduction

Instruction-set simulators are indispensable tools in the development of new architectures. They are used to validate an architecture design, a compiler design as well as to evaluate architectural design decisions during design space exploration. Running on a *host* machine, these tools mimic the behavior of an application program on a *target* machine. These simulators should be fast to handle the increasing complexity of processors, flexible to handle all features of applications and processors, e.g. runtime self modifying codes, multi mode processors; and retargetable to support a wide spectrum of architectures. Unfortunately none of the available simulation techniques focus on all of these issues at the same time.

Interpretive simulation is widely used due to its flexibility, despite its poor performance. In interpretive simulation, every time an instruction is fetched from memory, it is decoded and executed. By decoding all the instructions once prior to execution, the compiled simulation can significantly improve the simulation speed. However, the compiled simulation can not handle dynamic behavior of applications or processors that requires re-decoding of instructions during execution. As a result, compiled simulation can not handle many application domains, such as processors with multiple instruction set modes and self modifying programs.

A similar tradeoff between speed and retargetability exists in ISA simulators. Some of the retargetable simulators use a very general processor model and support a wide range of architectures but are slow, while others use some architectural or domain specific performance improvements but support only a limited range of processors. Although in the past years, performance has been the most important quality measure for the ISA simulators, retargetability is and additional concern, particularly in the area of the embedded systems. Today, an embedded application can be implemented on a variety of different architectures including microprocessors, DSPs and reconfigurable platforms. Besides, there are emerging architectures with combined features of classical architectures such as DSP, VLIW and Superscalar. For example, the TI C6x [19] family combines DSP and VLIW features and the Intel Itanium combines features of VLIW and superscalar architectures. To enable rapid design space exploration of such architectures, designers need a way of specifying a wide variety of processor-memory features and automatic generation of software toolkit including ISA simulators. A model is needed for capturing the features and techniques must be used for extracting the information and generating the simulator. While such a simulator must be fast and flexible, the model and the techniques must be general enough to support a wide spectrum of architectures. To the best of our knowledge, there is no published work on retargetable simulation that has focused on all these issues at the same time, while being able to deliver fast performance.

In this paper, we present a simulation framework that has the speed of compiled simulation and the flexibility of interpretive simulation while supporting many variations of architectures. We use the EXPRESSION ADL [13] to capture the architecture and generate the simulator from the ADL specification.

To achieve maximum retargetability, we have developed a generic instruction model coupled with a decoding technique that flexibly supports variations of instruction formats for widely differing contemporary processor architectures such as RISC, CISC, VLIW and variable length instruction set processors. To get high simulation performance, we have developed a technique called Instruction-Set Compiled Simulation (IS-CS). In IS-CS, instead of compiling the whole target program to a host binary, we compile each instruction to an optimized code for that instance of the instruction. This technique also enables us to use an interpretive simulation engine that can use pre-decoded optimized instructions as well as dynamically decoded instructions. Therefore the simulator uses the advantages of both techniques: the performance of compiled simulation and the flexibility of interpretive simulation.

The rest of the paper is organized as follows. Section 2 presents related work addressing ISA simulator generation techniques and distinguishes our approach. Section 3 outlines the retargetable simulation framework. It describes three key components of the framework: a generic instruction model, a decoding algorithm, and the instruction-set compiled simulation (IS-CS) technique. Section 4 presents simulation results using two contemporary processor architectures: ARM7 and SPARC. Section 5 concludes the paper.

2 Related Work

An extensive body of recent work has addressed instruction-set architecture simulation. The wide spectrum of today's instruction-set simulation techniques includes the most flexible but slow interpretive simulation and faster compiled simulation. Recent research addresses retargetability of instruction-set simulators using machine description languages.

Embra [15] and FastSim [14] simulators use dynamic binary translation and result caching to improve simulation performance. Embra provides the highest flexibility with maximum performance but is not retargetable and is restricted to the simulation of the MIPS R3000/R4000 architecture. It is inspired by Shade simulator [18] which uses a similar technique and can simulate the SPARC V8, V9 and MIPS instruction set at speeds of 3-10 times slower than native execution. SimpleScalar [21] is a popular interpretive simulator that supports a number of contemporary architectures but is not retargetable.

A fast and retargetable simulation technique is presented in [5]. It improves traditional static compiled simulation by aggressive utilization of the host machine resources. Such utilization is achieved by defining a low level code generation interface specialized for ISA simulation. This approach requires C descriptions that are based on the internal implementation details of the simulator rather than the specification of the target architecture.

Retargetable fast simulators based on an ADL have been proposed within the framework of FACILE [11], Sim-nML [16], ISDL [7], MIMOLA [17], and LISA ([3], [4]). The simulator generated from a FACILE description utilizes the Fast Forwarding technique to achieve reasonably high performance. All of these simulation approaches assumes that the program code is run-time static and have a limited retargetability. For example, Sim-nML only supports DSP processors while ISDL is mainly targeted at RISC machines. FLEXWARE Simulator [6] uses a VHDL model of a generic parameterizable model. SimC [12] is based on a machine description in ANSI C. It uses compiled simulation and has limited retargetability.

The published results of the LISA framework show successful retargetability for DSP and VLIW processors. The just-in-time cache compiled simulation (JIT-CCS) [2] technique, the closest to our approach, combines retargetability, flexibility and high simulation performance. The JIT-CCS performance improvement is gained by caching the decoded instruction information. This technique makes an assumption to get performance closer to compiled simulation: the number of repeatedly executed instructions should be very large such that 90% of the execution time is spent in 10% of the code. This assumption may not hold true for all real world applications. For example, the 176.gcc benchmark from SPEC CPU2000 violates this rule.

Our simulation framework supports a wide spectrum of processor architectures including RISC, DSP, VLIW, Superscalar and Hybrid architectures. In spite of being truly retargetable, our ISA simulator delivers up to 46% performance improvement over JIT-CCS. To have a fast, flexible and retargetable ISA simulator the following two questions must be answered:

First, *what limits the retargetability*? The existing ADL based approaches describe the instructions of an architecture based on a predefined model and then use a fixed decoding algorithm to decode target instruction binaries. This fixed decoding algorithm may not always work because different architectures use different decoding schemes for this purpose. This limitation can be solved by including necessary information of the instruction decoding algorithm in the specification. In our approach we use this information to extract the target decoding scheme.

Second, *how are performance and flexibility related*? The performance of a simulator depends on the overhead of simulating the program vs. executing it natively. To reduce this overhead, in a compiled simulation the program is decoded back into a source code with the same functionality and then, this source code is compiled and optimized on the host to get the best possible performance. In general, the flexibility of a simulator is defined by the granularity of portions of the program to which the decoding and optimizations are applied. Instruction level granularity is normally sufficient for flexible ISA simulators and therefore the performance optimizations should be applied to an instruction. In our simulator we generate optimized code for each instance of the instructions and execute them one by one. In this way, we can reinitiate the decoding of each instruction, if necessary.

3 Retargetable Simulation Framework

In a retargetable ISA simulation framework, the range of architectures that can be captured and the performance of the generated simulators depend on three issues: first, the model based on which the instructions are captured; second, the decoding algorithm that uses the instruction model to decode the input binary program; and third, the execution method of decoded instructions. These issues are equally important and ignoring any of them results in a simulator that is either very general but slow or very fast but restricted to some architecture domain. However, the instruction model significantly affects the complexity of decode and the quality of execution. We have developed a generic instruction model coupled with a simple decode algorithm that lead to an efficient and flexible execution of decoded instructions.

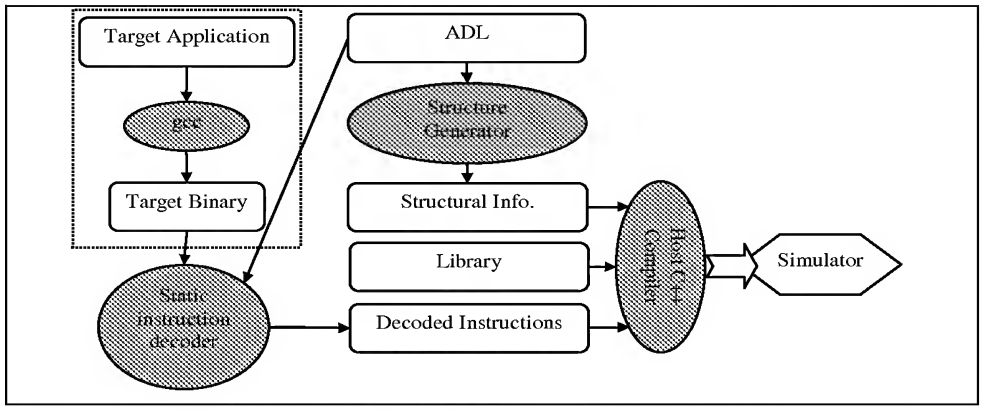

Figure 1- Generating the simulator from ADL

Figure 1 shows our retargetable simulation framework that gets the ADL (written in EXPRESSION) and the application program binary (compiled by *gcc*) and generates the simulator. The ADL captures behavior and structure of the target architecture. The behavioral part of the ADL is based on the generic instruction model, as described in Section 3.1, and is used by the *Static Instruction Decoder*. The structural information is used by the *Structure Generator*. Using the instruction specifications from ADL, the *Static Instruction Decoder* decodes the target program one instruction at a time, as described in Section 3.2. It then generates the optimized source code of the decoded instructions using IS-CS technique (Section 3.3) that is loaded in the instruction memory.

The S*tructure Generator* compiles the structural information of the ADL into components and objects that keep track of the state of the simulated processor. It generates proper source code for instantiating these components at run time.

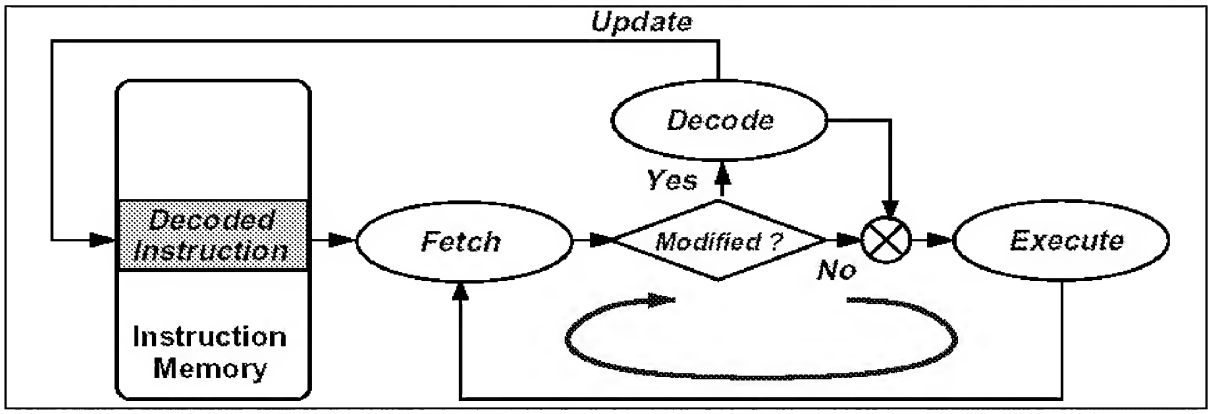

Figure 2- Instruction-Set Compiled Simulation Flow

The target independent components are described in the *Library*. This library is finally combined with the *Structural Information* and the *Decoded Instructions* and is compiled on the host machine to get the final ISA simulator. Figure 2 shows the flow of the simulation engine. This engine fetches the decoded instructions from the instruction memory and executes them. If the simulator detects that the program code of a previously executed address has changed it initiates a re-decoding and then updates the instruction memory. This technique combines the performance of traditional compiled simulators with the flexibility of interpretive simulation and achieves the highest possible performance due to two reasons. First, the time consuming instruction decoding process is moved to compile time. Second, we use a novel instruction abstraction technique to generate aggressively optimized decoded instructions that further improves simulation performance as described in the section 3.3.

In the remainder of this section, we describe the generic instruction model, followed by how we capture instructions in the ADL using the generic model. Then, we explain how the decoding algorithm decodes the program binary using the description of instructions in the ADL. Finally, we show how the IS-CS technique achieves its speed and flexibility using the optimized decoded instructions.

3.1 Generic Instruction Model

A major challenge in retargetable simulation is the ability to capture a wide variety of instructions. In this section we propose an instruction model that is generic enough to capture variations of instruction formats of contemporary processors such as RISC, CISC, VLIW and variable length instruction set processors. As an illustrative example, we use integer arithmetic instructions of the Sparc V7 processor to explain this model.

Example 1: Sparc V7 [22] is a single-issue processor with 32-bit instruction. The integer-arithmetic instructions, *IntegerOps* (as shown below), are a subset of the instruction set that perform certain arithmetic operation on two source operands and write the result to the destination operand. So the behavior of these instructions can be defined as: $dest=f_{opcode}(src1, src2)$. The destination and the first source operand are always a register, but the second source operand can be a register or an immediate integer constant. This subset of instructions is distinguished from the others by the following bit mask:

IntergerOps: <opcode dest src1 src2> Bitmask:

10xxxxx0	xxxxxxxx	xxxxxxxx	xxxxxxxx

A bit mask is a string of '1', '0' and 'x' symbols and it matches a bit pattern of the binary instruction if and only if for each '1' or '0' in the mask, the binary instruction has a 1 or a 0 value in the corresponding position respectively. The 'x' symbol matches with both 1 and 0 values.

In this model, an *instruction* of a processor is composed of a series of *slots*, $I =< sl_0, sl_1, ... >$, and each slot contains only one *operation* from a subset of operations. All the operations in an instruction execute in parallel. For example, in the TI C6x architecture a VLIW instruction has 8 slots; hence it can have up to 8 concurrent operations. Each operation is distinguished by a mask pattern. Therefore, each slot (sl_i) contains a set of operation-mask pairs (op_i, m_i) and is defined in the following format. The length of an operation is equal to the length of mask pattern.

$$sl_i =< (op_i^0, m_i^0) \mid (op_i^1, m_i^1) \mid ... >$$

An *operation class* refers to a set of similar operations in the instruction set that can appear in the same instruction slot. The previous slot description can be rewritten using an operation class *clOps*: $sl_i =< (clOps_i, m_i) >$. An instruction in the Sparc V7 processor of Example 1 can have only one slot (one operation). For example, integer arithmetic instructions in Sparc V7 can be grouped in a class (IntegerOps) as shown below:

$$I_{SPARC} = <(IntegerOps, 10xx\text{-}xxx0\ xxxx\text{-}xxxx\ xxxx\text{-}xxxx\ xxxx\text{-}xxxx) \mid ... >$$

An operation class is composed of a set of *symbols* and an *expression* that describes the behavior of the operation class in terms of the values of its symbols. For example, the operation class in Example 1 has four symbols: *opcode*, *dest*, *src1* and *src2*. The *expression* for this example will be: $dest = f_{opcode}(src1, src2)$. Each symbol may have a different *type* depending on the bit pattern of the operation instance in the program. For example, the possible types for src2 symbol in Example 1 are register and immediate integer. The value of a symbol depends on its type and can be static or dynamic. For example, the value of a register symbol is dynamic and is known only at run time, whereas the value of an immediate integer symbol is static and is known at compile time. In general, each symbol in an operation has a possible set of types. A general operation class is then defined as:

clOps =< $(s_0, T_0), (s_1, T_1), \ldots) \mid exp(s_0, s_1, \ldots) >$, where (s_i, T_i) are *(symbol, type)* pairs and $exp(s_0, s_1, \ldots)$ is the behavior of the operations based on the values of the symbols.

The type of a symbol can be defined as a register ($\in$ *Registers*) or an immediate constant ($\in$ *Constants*) or can be based on certain *micro-operations* ($\in$ *Operations*). For example, a data processing instruction in ARM (e.g., add) uses shift (micro-operation) to compute the second source operand, known as ShifterOperand. Each possible type of a symbol is coupled with a mask pattern that determines what bits in that operation must be checked to find out the actual type of the corresponding symbol. In general, possible types of a symbol are defined as:

$$T = \{(t, m) \mid t \in \text{Operations} \cup \text{Registers} \cup \text{Constants}, m \in (1 \mid 0 \mid x)^*\}$$

For example, the opcode symbol in Example 1 can be any of valid integer arithmetic operations and can be described as:

```
OpTypes = {
    (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
    (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
    ...
}
```

Usually the registers of a processor are divided into groups known as register classes. For example, the SPARC processor has three register classes: integer registers, floating point registers and control registers. The actual register in a processor is defined by its class and its index. The index of a register in an instruction is defined by extracting a slice of the instruction bit pattern and interpreting it as an unsigned integer. An instruction can also use a specific register with a fixed index as in a branch instruction that update the program counter. In general a register is defined by r = [regClass, i, j]|[regClass, index] where *i* and *j* define the boundary of index bit slice in the instruction. For example, the *dest* symbol (in Example 1) is 25^{th} to 29^{th} bits in the instruction, and is an integer register. Its type can be described as: *DestType= (IntegerRegClass, 29, 25).*

```
SPARCInst = $
    (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) | ...
$;
IntegerOp = <
    (opcode, OpTypes), (dest, DestType), (src1, Src1Type), (src2, Src2Type)
    | {
        dest = opcode(src1, src2);
    }
>;
OpTypes = {
        (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
        (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
        (Or , xxxx-xxxx 0010-xxxx xxxx-xxxx xxxx-xxxx),
        (And, xxxx-xxxx 0001-xxxx xxxx-xxxx xxxx-xxxx),
        (Xor, xxxx-xxxx 0011-xxxx xxxx-xxxx xxxx-xxxx),
        ...
};
DestType = [IntegerRegClass, 29, 25];
Src1Type = [IntegerRegClass, 18, 14];
Src2Type = {
        ([IntegerRegClass,4,0], xxxx-xxxx xxxx-xxxx xx0x-xxxx xxxx-xxxx),
        (#int,12,0#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Figure 3- Integer arithmetic instcutions in SPARC

Similarly a portion of an instruction may be considered as a constant. For example, one bit in an instruction can be equivalent to a Boolean type or a set of bits can make an integer immediate. It is also possible to have constants with fixed values in the instructions. In general a constant type is defined by c =# type,i, j# | # type, value# where *i* and *j* show the bit positions of the constant and *type* is a scalar type such as integer, Boolean, float, etc.

Figure 3 shows the complete description of integer-arithmetic instructions in SPARC processor (Example 1). This simple example also shows how similar operations can be grouped together and described easily. We are able to describe instruction sets of a wide range of architectures using our generic instruction model. Figure 4 describes how to capture instruction set of the ARM processor using the instruction model.

The ARM instructions are 32-bit wide and all are conditional. In data-processing instructions, if the condition is true, some arithmetic operation is performed on the two source operands and the result is written in the destination operand. The destination and the first source operand are always registers. The second source operand, called ShifterOperand, has three fields: shift operand (register/immediate), shift operation (5 types) and shift value (register/immediate). The shift value shows the number of shifts that must be performed on the shift operand by the specified shift operation. For example, the "*ADD r1, r2, #10 sl r3*" is equivalent to "*r1=r2+(10 << r3)*" expression. If indicated in the instruction opcode, the flag bits (Z, N, C, and V) are updated.

We defined a set of macros that can be used for compact description. For example, to avoid long mask strings with many *don't care* bits, the *mask* macro can be used. This macro gets the length of a mask, a bit position and a string. It then generates a bit mask with the specified size and copies the string at the corresponding bit position and fills the rest of the bit mask with 'x' symbols. For example *mask(8, 2,"10")* generates an 8 bit mask that has a '10' at position 2 i.e. xxxx-x10x.

```
ARMInst = $
      (DPOperation, xxxx-001x xxxx-xxxx xxxx-xxxx xxxx-xxxx) |
      (DPOperation, xxxx-000x xxxx-xxxx xxxx-xxxx xxx0-xxxx) |
      (DPOperation, xxxx-000x xxxx-xxxx xxxx-xxxx 0xx1-xxxx) |
      …
$;
DPOperation = <
      (cond, Conditions), (opcode, Operations), (dest, [intReg,15,12]), (src1, [intReg,19,16]), (src2, ShifterOperand),
      (updateFlag, {(true, mask(32, 20, "1"), (false, mask(32, 20, "0")})
   | {
      if (cond()) {
         dest = opcode( src1, src2);
         if (updateFlags) {/*Update flags*/}
      }
   }
>;
Conditions = {
      (Equal, mask(32, 31, "0000"), (NotEqual, mask(32, 31, "0001"), (CarrySet, mask(32, 31, "0010"),
      (CarryClear, mask(32, 31, "0011"), …, (Always, mask(32, 31, "1110"), (Never, mask(32, 31, "1111")
};
Operations = {
      (And, mask(32, 24, "0000"), (XOr, mask(32, 24, "0001"), (Sub, mask(32, 24, "0010"), (Add, mask(32, 24, "0100"), …
};
ShifterOperand = <
      (op, {([intReg,11,8], mask(32,4,"0")), (#int,11,7#, mask(32,7,"0xx1"))}),
      (sh, {(ShiftLeft, mask(32,6,"00)), (ShiftRight, mask(32,6,"01)), …}),
      (val, {([intReg,3,0], mask(32,25,"0")), (#int,7,0#, mask(32,25,"1"))})
   | {  sh(op, val)  }
>;
```

Figure 4- Data processing instructions in ARM

To see how compact and efficient this model is, consider the number of different instructions that this small description in Figure 4 defines. The data processing instructions in ARM processor can have 16 conditions and 16 operations. The shifter operand supports 5 shift operations and the parameters can be register or immediate (5x2x2). Each instruction may or may not update the flag bits (2). Therefore there will be 16x16x(5x2x2)x2=10240 possible formats in this category. In the next sections, we show how all these possibilities are explored for generating an optimized code for each type of instruction. In this model, instructions that have similar format are grouped together into one class. Most of the time this information is readily available from the instruction set architecture manual. For example, we defined six instruction classes for the ARM processor viz., Data Processing, Branch, LoadStore, Multiply, Multiple LoadStore, Software Interrupt, and Swap.

In this section, we have demonstrated two key features of our instruction model: first, it is generic enough to capture architectures with varied instruction sets; second, it captures the instructions efficiently by allowing instruction grouping.

3.2 Generic Instruction Decoder

A key requirement in a retargetable simulation framework is the ability to automatically decode application binaries of different processors architectures. This necessitates a generic decoding technique that can decode the application binaries based on instruction specifications. In this section we propose a generic instruction decoding technique that is customizable depending on the instruction specifications captured through our generic instruction model.

Algorithm 1: *StaticInstructionDecoder*
Input: Target Program Binary *Appl*, Instruction Specifications *InstSpec;*
Output: Decoded Program *DecodedOperations*;
Begin
    Addr = Address of first instruction in *Appl;*
  While (*Appl* not processed completely)
    BinStream = Binary stream in *Appl* starting at *Addr*;
    (*Exp*, *AddrIncrement*) = DecodeOperation (*BinStream*, *InstSpec*);
    DecodedOperations = Append (*Exp*, *Addr*, *DecodedOperations*);
    Addr = *Addr* + *AddrIncrement*;
  EndWhile;
  return *DecodedOperations* ;
End;

Algorithm 1 describes how *Static Instruction Decoder* of Figure 1 works. This algorithm accepts the target program binary and the instruction specification as inputs and generates a source file containing decoded instructions as output. Iterating on the input binary stream, it finds an operation, decodes it using Algorithm 2, and adds the decoded operation to the output source file. Algorithm 2 also returns the length of the current operation that is used to determine the beginning of the next operation.

Algorithm 2: *DecodeOperation*
Input: Binary Stream *BinStream*, Specifications *Spec*;
Output: Decoded Expression *Exp*, Integer *DecodedStreamSize*;
Begin
  (*OpDesc*, *OpMask*) = findMatchingPair(*Spec*, *BinStream*);
  OpBinary = initial part of *BinStream* whose length is equal to *OpMask*;
  Exp = the expression part of *OpDesc*;
  ForEach pair of (*s*, *T*) in the *OpDesc*
    Find *t* in *T* whose mask matches the *OpBinary*;
    v = ValueOf(*t*, *OpBinary*);
    Replace *s* with *v* in *Exp*;
  EndFor
  return (*Exp* , size(*OpBianry*));
End;

Algorithm 2 gets a binary stream and a set of specifications containing operation or micro-operation classes. The binary stream is compared with the elements of the specification to find the specification-mask pair that matches with the beginning of the stream. The length of the matched mask defines the length of the operation that must be decoded. The types of symbols are determined by comparing their masks with the binary stream. Finally, using the symbol types, all symbols are replaced with their values in the expression part of the corresponding specification. The resulting expression is the behavior of the operation. This behavior and the length of the decoded operation are produced as outputs.

Consider the following SPARC Add operation example and its binary pattern:

	31	23	15	7
Add g1, #10, g2	1000-0100	0000-0000	0110-0000	0000-1010

In the first line of Algorithm 2, the (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) pair matches with the instruction binary. This means that the *IntegerOps* operation class matches this operation. It calls Algorithm 3 to decode the symbols of *IntegerOps* viz. *opcode, dest, src1, src2.*

```
Algorithm 3: ValueOf
Input: Type t, Operation Binary OpBinary;
Output: Extracted Value extValue;
Begin
   Switch (t)
      case #type, value#: extValue = (type) value; endcase
      case #type, i, j#: extValue = (type) OpBinary[i:j]; endcase
      case [regClass, index]: extValue = REGS[regClass][index]; endcase
      case [regClass, i, j]: extValue = REGS[regClass][ OpBinary[i:j]]; endcase
      case Operation Spec:
            (extValue, tmp) = DecodeOperation(OpBinary, t);
      endcase
   EndSwitch;
   return extValue;
End;
```

Algorithm 3 gets a symbol type and an operation binary (*OpBinary*), and returns the actual value of the corresponding symbol. If the type itself is a micro-operation specification, the decode algorithm (Algorithm 2) is called again and the result is returned. If the type is not a fixed constant (register), the value is calculated by interpreting the proper portion of the operation binary (*OpBinary[i:j]*) as a constant (register index).

In the previous example, the 4 symbols (*opcode, dest, src1, src2*) are decoded using Algorithm 3. Symbol *opcode*'s type is *OpTypes* in which the mask pattern of *Add* matches the operation pattern. So the value of *opcode* is *Add* function. Symbol *dest*'s type is *DestType* which is a register type. It is an integer register whose index is bits 25^{th} to 29^{th} (00010), i.e. 2. Similarly, symbol *src1*'s type is *Src1Type* which is a register type. It is a register integer whose index is bits 14^{th} to 18^{th} (00001), i.e. 1. Likewise, symbol *src2*'s type is *Src2Type* in which the mask pattern of #int,12,0# matches the operation binary pattern. This means that the bits 0^{th} to 12^{th} (0000000001010) must be interpreted as an integer, i.e. 10. By replacing these values in the expression part of the *IntegerOps* the final behavior of the operation would be: g2 = Add(g1, 10); which means g2 = g1 + 10.

The complexity of the decoding algorithm is $O(n*m*log_2 m)$, where n is the number of operations in the input binary program, m is the number of operations in the architecture instruction set.

3.3 IS-Compiled Simulation (IS-CS)

We developed the instruction set compiled simulation (IS-CS) technique with the intention of combining the full flexibility of interpretive simulation with the speed of the compiled principle. The basic idea is to move the time-consuming instruction decoding to compile time as shown in Figure 5. The application program, written in C/C++, is

compiled using the gcc compiler configured to generate binary for the target machine. The instruction decoder decodes one binary instruction at a time to generate the decoded program for the input application. The decoded program is compiled by C++ compiler and linked with the simulation library to generate the simulator. The simulator recognizes if the previously decoded instruction has changed and initiates re-decoding of the modified instruction. If any instruction is modified during execution and subsequently re-decoded, the location in instruction memory is updated with the re-decoded instruction.

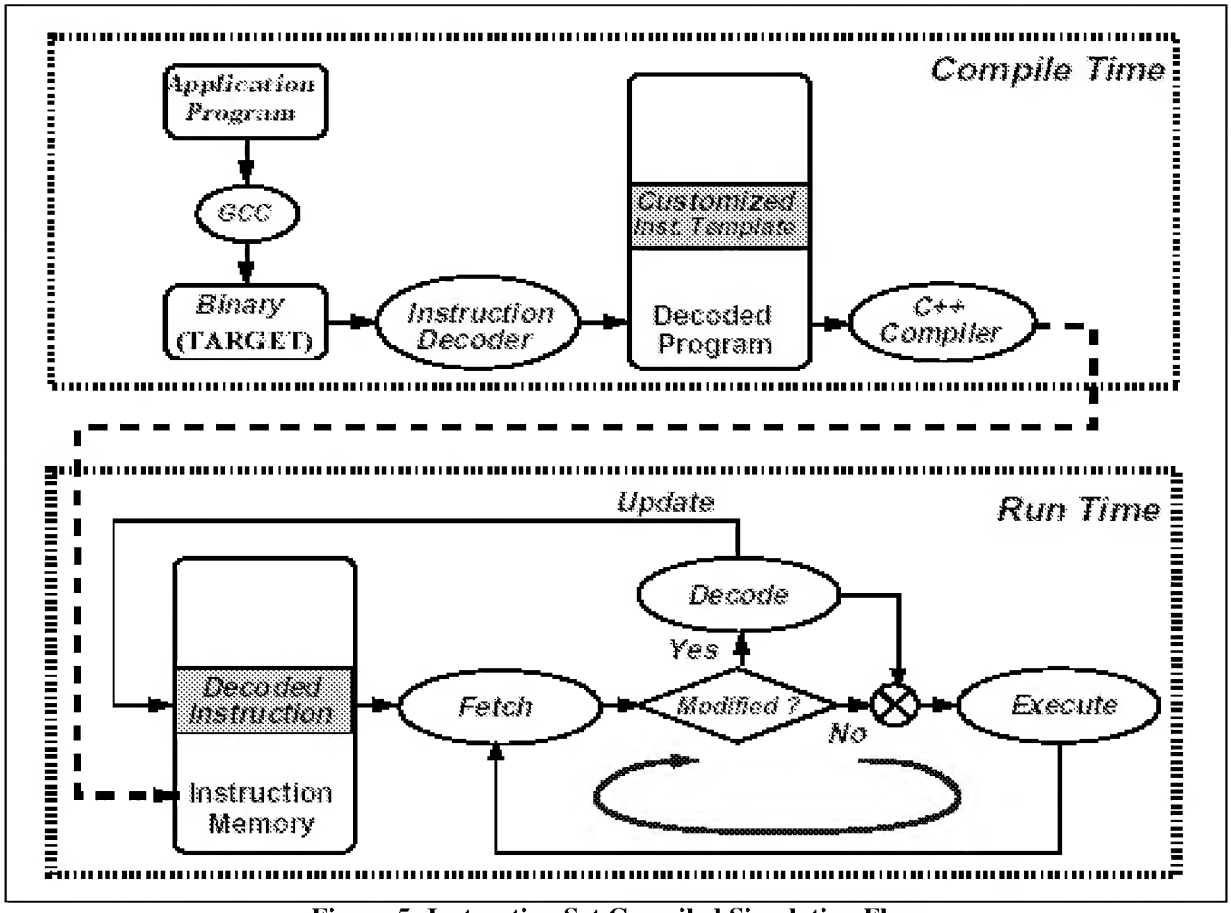

Figure 5- Instruction Set Compiled Simulation Flow

To improve the simulation speed we use a novel instruction abstraction technique that generates optimized decoded instructions as described in Section 3.1. As a result the computation during run-time is minimized.

In traditional interpretive simulation (e.g., Simplescalar [21]) the decoding and execution of binary instructions are done using a single monolithic function. This function has many if-then-else and switch/case statements that perform certain activities based on bit patterns of opcode, operands, addressing modes etc. In advanced interpretive simulation (e.g., LISA [2]) the binary instruction is decoded and the decoded instruction contains pointers to specific functions. There are many variations of these two methods based on efficiency of decode, complexity of implementation, and performance of execution. However, none of these techniques exploit the fact that a certain class of instructions may have a constant value for a particular field of the instruction. For example, a majority of the ARM instructions execute unconditionally (condition field has value always). It is a waste of time to check the condition for such instructions every time they are executed.

Clearly, when certain input values are known for a class of instructions, the partial evaluation [13] technique can be applied. The effect of partial evaluation is to specialize a program with part of its input to get a faster version of the same

program. To take advantage of such situations we need to have separate functions for each and every possible format of instructions so that the function could be optimized by the compiler at compile time and produce the best performance at run time. Unfortunately, this is not feasible in practice. For example, consider the ARM data processing instructions. It can have 16 conditions, 16 operations, an update flag (true/false), and one destination followed by two source operands. The second source operand, called shifter operand, has three fields: operand type (reg/imm), shift options(5 types) and shift value (reg/imm). In total, the ARM data processing instructions have 16 x 16 x 2 x 2 x 5 x 2 = 10240 possible formats.

Our solution to this problem is to define instruction classes, where each class contains instructions with similar formats. Most of the time this information is readily available from the instruction set architecture manual. For example, we defined six instruction classes for the ARM processor viz., Data Processing, Branch, LoadStore, Multiply, Multiple Load-Store, Software Interrupt, and Swap. Next, we define a set of masks for each instruction class. The mask consists of '0', '1' and 'x' symbols. A '0' ('1') symbol in the mask matches with a '0' ('1') in binary pattern of the instruction at the same bit position. An 'x' symbol matches with both '0' and '1'. For example, the masks for the data processing instructions are shown below: "xxxx-001x xxxx-xxxx xxxx-xxxx xxxx-xxxx" "xxxx-000x xxxx-xxxx xxxx-xxxx xxx0-xxxx" "xxxx-000x xxxx-xxxx xxxx-xxxx 0xx1-xxxx" We use C++ templates to implement the functionality for each class of instructions. For example, the pseudo code for the data processing template is shown below. The template has four parameters viz., condition, operation, update flag, and shifter operand. The shifter operand is a template having three parameters viz., operand type, shift options and shift value.

Example 1: Template for Data Processing Instructions

```
template <class Cond, class Op, class Flag, class SftOper>
  class DataProcessing :
  {
    SftOper _sftOperand;
    Reg _dest, _src1;
  public:
    .......
    virtual void execute()
    {
    if (Cond::execute())
    {
      _dest = Op::execute(_src1, _sftOperand.getValue());
      if (Flag::execute())
      {
        // Update Flags
      }
    }
  }
};
```

We illustrate the power of our technique to generate an optimized decoded instruction using a single data processing instruction. We show the binary as well as the assembly of the instruction below.

```
Binary:    1110|000|0100|0|0010|0001|01010|00|0|0011
           (cond|000| op |S| Rn | Rd |shift immed|shift|0|Rm)
Assembly: ADD r1, r2, r3 LSL #10
           (op{<cond>}{S} Rd, Rn, Rm shift #<immed>)
```

The DetermineTemplate function returns the DataProcessing template (shown in Example 1) for this binary instruction. The CustomizeTemplate function generates the following customized template for the execute function.

```
void DataProcessing<Always, Add, False,
SftOper<Reg, ShiftLeft, Imm>>::execute()
{
```

```
  if (Always::execute()) {
    _dest = Add::execute(_src1, _sftOperand.getValue());
    if (False::execute()) {
      // Update Flags
      ...
    }
  }
}
```

After compilation using a C++ compiler, several optimizations occur on the execute() function. The Always::execute() function call is evaluated to true. Hence, the check is removed. Similarly, the function call False::execute() is evaluated to false. As a result the branch and the statements inside it are removed by the compiler. Finally, the two function calls Add::execute(), and sftOperand.getValue() get inlined as well. Consequently, the execute() function gets optimized into one single statement as shown below:

```
void DataProcessing<..skipped..>::execute() {
  _dest = _src1 + _sftOperand._operand << 10;
}
```

Furthermore, in many ARM instructions, the shifter operand is a simple register or immediate. Therefore, the shift operation is actually a no shift operation. Although the manual says that the case is equivalent to shift left zero, we use a no shift operation that enables further optimization. In this way, an instruction similar to the above example would have only one operation in its execute() method.

Similarly, Figure 6 shows the code generation process for the Sparc description shown in Figure 3.

```
/* generated template for integer arithmetic operations of Sparc*/
template<class OpTypes, class DestType, class Src1Type, class Src2Type>
class IntegerOps
{
      DestType dest;  Src1Type src1;  Src2Type src2;
  public:
      virtual void execute() { dest = OpTypes::f(src1, src2); }
      …
};

/* customized template for Add g1, #10, g2 instruction*/
void IntegerOps<Add, Register, Register, int>::execute(){
  REGS[IntegerRegClass][dest] = Add:f(REGS[IntegerRegClass][src1], src2);
}

/* optimized template for  Add g1, #10, g2 instruction*/
void IntegerOps<Add, Register, Register, int>::execute(){
  REGS[IntegerRegClass][dest] = REGS[IntegerRegClass][ src1] + src2;
}
```

Figure 6- Code generation for a Sparc instruction

4 Experiments

In order to evaluate the applicability of our framework to generate fast, flexible and retargetable simulator, we performed several experiments using various processor models. In this section, we present simulation results using two contemporary processors: ARM7 [20] and SPARC [22] to demonstrate the usefulness of our approach.

4.1 Experimental Setup

The ARM7 processor is a RISC machine with fairly complex instruction set. We used *arm-linux-gcc* for generating target binaries for ARM7 and validated the generated simulator by comparing traces with Simplescalar-arm [21] simulator.

The Sparc V7 is a high performance RISC processor with 32-bit instructions. We used *gcc3.1* running on Solaris to generate the target binaries for Sparc and validated the generated simulator by comparing traces with *Shade* [18] simulator.

Performance results for simulators were obtained using a 1 GHz PentiumIII with 256 MB RAM running Windows 2000 Professional. The generated simulator code was compiled using the Microsoft Visual Studio .NET compiler with all optimization enabled. We also used a 400 MHz UltraSparc-II with 1 GB RAM and gcc3.1 with all optimizations to compare the performance of the simulator against running the program natively. In both cases, the same C++ compiler was used for compiling the decoded program.

We have used benchmarks from SPEC 95 and DSP domains. In this section we show the results using three application programs: *adpcm* from DSP benchmark suite, *099.go* and *129.compress* from SPEC 95.

4.2 Results

Figure 7 shows the simulation performance of our technique using the ARM7 model. The first bar shows the simulation performance with dynamic checking of program modification. The second bar presents the performance without checking run time changes.

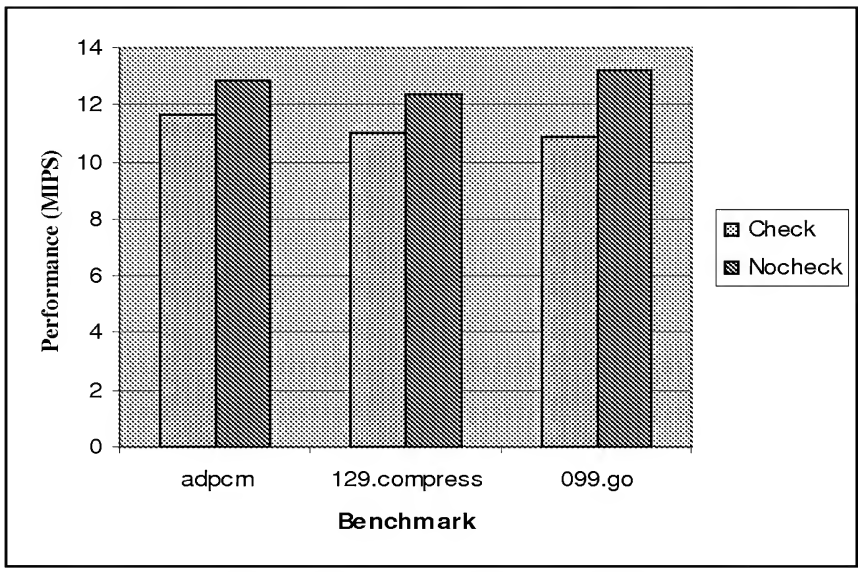

Figure 7- Simulation Results – ARM7

Our simulation performance is superior to all variations of interpretive simulation techniques published in the literature. To the best of our knowledge the best performance of a simulator having the flexibility of interpretive simulation has been JIT-CCS [2]. The JIT-CCS technique could achieve a performance up to 8 MIPS on an Athlon at 1.2 GHz with 768 MB RAM for *adpcm* benchmark on ARM7 simulator. Our simulation technique delivers up to 46% performance improvement (11.7 MIPS with dynamic checking) using a slower machine (P3 1.0 GHz with 256 MB RAM). The performance improvement is even higher (12.9 MIPS) when dynamic checking is disabled. There are two reasons for the superior performance of our technique: moving the time consuming decoding out of the execution loop, and generating aggressively optimized code for each instruction.

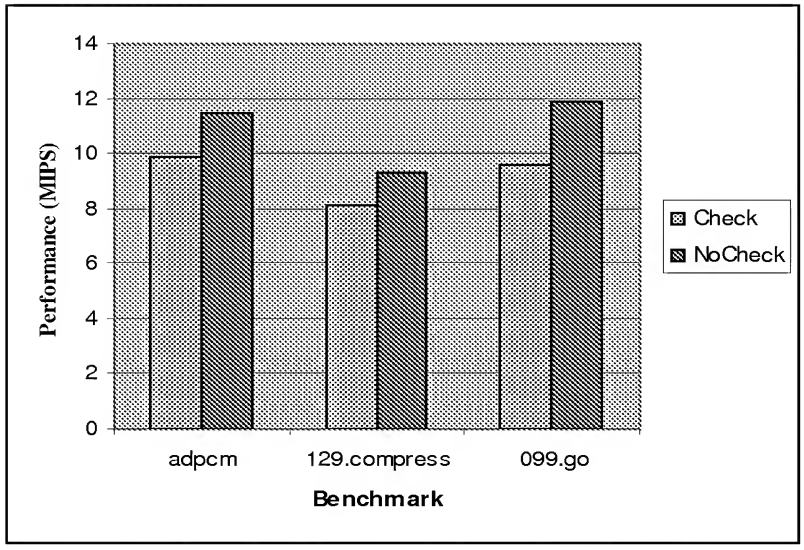

Figure 8- Simulation Results - Sparc

Figure 8 shows the simulation performance of our technique using Sparc model. The first bar shows the simulation performance with dynamic checking of program modification. The second bar presents the performance with checking disabled.

Note that in IS-CS, each instruction is optimized and hence depending on their complexity, different instructions have different performance. For example, in both Sparc and ARM architectures, the most complex instructions are those that are used for context switching. These instructions are used to save or restore a group of registers. The reason for the poor performance of the compress benchmark in both simulators is mainly because of too many Save and Restore operations.

Also note that, the overall performance of ARM simulator is slightly better than that of Sparc. ARM instructions are more complex and in most cases are equivalent to more than one Sparc instruction. Therefore optimizing one ARM instruction is equivalent to optimizing multiple instructions in Sparc.

Table 1 compares the simulation results of our Sparc simulator (with dynamic checking disabled) against native execution. We ran both the simulator and the application program on a 400 MHz UltraSparc-II with 1 GB RAM. Our simulator is approximately 100 times slower than the native execution. The poor performance for *compress* benchmark is due to the simulation of too many save and restore instructions.

Table 1- Native Execution Vs Simulation Performance

Benchmarks	Native Execution$_{(MIPS)}$	Simulation$_{(MIPS)}$	Ratio
adpcm	390.71	3.9	100
compress	487.12	2.9	167
go	202.07	4.0	50

We have demonstrated that our framework can generate retargetable simulators that deliver the performance of compiled simulation while maintaining the flexibility of interpretive simulation. Our simulation technique delivers better performance than any simulators in this category, as demonstrated in this section.

5 Summary

In this paper, we presented a framework for generating fast, flexible, and retargetable ISA simulator. We proposed a generic instruction model as well as a generic decode algorithm that can specify and decode many variations of instructions. Besides, due to the simple interpretive simulation engine and optimized pre-decoded instructions, our

instruction set compiled simulation (IS-CS) technique achieves the performance of compiled simulation while maintaining the flexibility of interpretive simulation. The IS-CS technique achieves its superior performance for two reasons: moving time-consuming decode to compile time, and using templates to produce aggressively optimized code for each instance of instructions. We demonstrated performance improvement of up to 46% over the best published results on an ARM7 model. Our framework uses the EXPRESSION ADL to capture the processor architecture and generate retargetable simulator. The generic instruction model coupled with decoding algorithm form the backbone of our retargetable framework. Future work will concentrate on using this framework for modeling other real world architectures.

6 Acknowledgements

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge Dan Nicolaescu, Radu Cornea and the members of the ACES laboratory for their inputs, .

7 Reference

[1] V. Zivojnovic et al. *LISA - machine description language and generic machine model for HW/SW co-design*. In IEEEWorkshop on VLSI Signal Processing, 1996.

[2] A. Nohl et al. *A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation*. DAC,2002.

[3] S. Pees et al. *Retargeting of Compiled Simulators for Digital Signal Processors using a Machine Description Language*. DATE, 2000.

[4] G. Braun et al. *Using Static Scheduling Techniques for the Retargeting of High Speed, Compiled Simulators for Embedded Processors from an Abstract Machine Description*. ISSS, 2001.

[5] J. Zhu et al. *A Retargetable, Ultra-fast Instruction Set Simulator*. DATE, 1999.

[6] P. Paulin et al. FlexWare: *A flexible firmware development environment for embedded systems*. In Proc. Dagstuhl Code GenerationWorkshop, 1994.

[7] G. Hadjiyiannis et al. *ISDL: An instruction set description language for retargetability*. In Proc. DAC, 1997.

[8] M. Freericks. The *nML machine description formalism*. Technical Report TR SMIMP/DIST/08, TU Berlin CS Dept., 1993.

[9] Trimaran Release: http://www.trimaran.org. The MDES User Manual, 1998.

[10] R. Leupers, J. Elste, and B. Landwehr. *Generation of interpretive and compiled instruction set simulators*. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 1999.

[11] E. Schnarr, M. Hill, and J. Larus. *Facile: A language and compiler for high-performance processor simulators*. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Utah, USA, Jun. 2001.

[12] F. Engel, J. N¨uhrenberg, and G. Fettweis. *A generic tool set for application specific processor architectures*. In Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES), pages 126–130, San Diego, CA USA, May 2000.

[13] A. Halambi et al. *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability*. DATE, 1999.

[14] E. Schnarr et al. *Fast Out-of-Order Processor Simulation using Memoization*. PLDI, 1998.

[15] E. Witchel et al. *Embra: Fast and Flexible Machine Simulation*. MMCS, 1996.

[16] M. Hartoog et al. *Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign*. DAC, 1997.

[17] R. Leupers et al. *Generation of Interpretive and Compiled Instruction Set Simulators*. DAC, 1999.

[18] Robert. F. Cmelik, David Keppel. *Shade: A fast instruction set simulator for execution profiling*. Proceedings of 1994 ACM SIGMETTRICS Conference on Measurment and Modeling of computer systems, Philadelphia, 1996.

[19] Texas Instruments, TMS320C6201 CPU and Instruction Set Reference Guide, 1998.

[20] The ARM7 User Manual, http://www.arm.com.

[21] Simplescalar Home page: http://www.simplescalar.com

[22] Sparc Version 7 Instruction set manual: http://www.atmel.com/dyn/resources/prod_documents/doc3b8b88df7a415.pdf